PAGING

Andrew Quinn

Learning Objectives:

1. How do we translate addresses when using paging?
2. How does demand paing work?
3. What tradeoffs are there for determining page size?

Announcements:

1. Have you started working on Project 2?

We saw that segmentation gave us *many* of the things that we want from a virtual memory subsystem. But, external fragmentation is a *major* issue. If you pay for all that memory, you *really* want to be able to use it [1]!

The primary goal of paging is to solve the external fragmentation problem. Serendipitously, paging makes it much easier to solve expensiveness by letting virtual memory expand beyond the confines of a single machine. But, as the economists say, "there is no free lunch". Paging comes with some costs; we'll mostly turn to hardware to solve them.

PAGING

Rather than split memory into variable sized regions, or segments, why don't we instead split our memory spaces (both virtual and physical) into *fixed sized chunks*?

- We'll call the chunks in a process' virtual memory space a *page*
- We'll call the chunks in the system's physical memory space a *page frame*, or just a "frame".
- Any page can reside in any frame—no need for continuity—so no more external fragmentation...at least at the OS level.
- We'll call the mapping from a process' pages into the frames where they actually reside a *page table*. Page tables are large enough to be stored in memory (almost always).

ADDRESS TRANSLATION

To actually translate an virtual address into a physical one, we first determine the page in which the virtual address resides. Then, we use the page table to determine the physical frame that contains the virtual page. Finally, we create the physical address by adding the offset to the physical frame. If we try to use a virtual page that is invalid, then the hardware will raise a *page fault* to tell the operating system that something bad has happened.

Here's a cool trick: if you make your page size a power of two, then the page containing a virtual address is always some of the high-order bits of the virtual address, and the offset is always some low-order bits. For example, suppose 32-bit addresses and 8-byte pages. The

[1]As we mentioned in class, external fragmentation is still a major problem. From malloc to memory allocation for virtual machines, we cannot seem to get away from it

first 29 bits of a virtual address would encode the page index, while the final 3 bits would encode the offset into the page. If your pages are of size 2^k and your addresses are of size n, then the offset is the low-order k bits of the virtual addresses, while the page index is the $n - k$ high order bits.

Recall that we want hardware to do translation so that we have an efficient translation scheme. More on this later.

DEMAND PAGING—PROVIDING EXPENSIVENESS

When we actually execute an instruction, like a load, we need virtual pages to map to physical ones. But, what about when an application is not using memory? Why do virtual pages need to map to physical frames? We don't!

So, the operating system allows pages to be located "somewhere else". For this class, that somewhere else will be in *swap* space on disk. The operating system splits a special disk partition into page/frame-sized chunks. Your book calls these chunks, *blocks*. I think this is a mistake because a block is a different abstraction, so I'll call them *swap frame*s. Then, the page table will include additional metadata. We add extra state, called *location*, set to either mem or swap, and some columns to track the swap frame that each virtual page that holds a swap frame.

Let's say that an application needs to use memory on a page that is currently located in swap. Who manages that? There's basically two options. You could have hardware "page in" the page. But, swapping requires reading from disk, which is slow, and the hardware has no way of context switching to hide that cost. So, systems instead have the operating system manage swapping.

Here's how it works. If a page is located in memory, then the hardware performs the address translation using the page table as described above. However, if the page is *not* located in memory, then the hardware raises a page fault, in the same way that it does for pages that have no valid mapping.

The operating system then "pages-in" the demanded page. Note that the operating system may have to "swap out" some other page. So, it does the following:

1. Are there any empty frames?
2. If no, evict a page from a frame by writing it to swap. Assign frame to be the evicted frame.
3. If yes, chose frame to be any empty frame.
4. Copy the page from swap into frame.
5. Update the page table.

Because the hardware behavior is the same when a page is invalid and when a page is in not resident, most *hardware page tables*, or the page tables that hardware uses for translation, are different from the page tables that we've seen so far. In other words: there's usually two separate versions of a page table: one for the OS and one for the hardware.

This discussion has centered on applications using virtual memory to map between pages and frames. But, the kernel can also page its own memory.

PAGE SIZE TRADEOFFS

How big should our pages be? Its a tradeoff between internal fragmentation and page table size.

Let's make it concrete: suppose that we had 1024 byte pages. Then, your page table would be "one order" smaller than your address space: 4GB address space requires 4MB page tables, 256 TB address space (as is allowed in 48-bits) requires 256 GB page tables.

So, we want big pages right! Not so fast: if you make pages very large, then you increase *internal fragmentation*. Even well behaved applications tend to use lots of the virtual address space so that their stack, heap, and code are in separate regions. They're likely to have some partially used pages for each of these regions, and so there will be a lot of waste if you make pages quite large.

In practice, most operating systems have converged around a 4KB page. There's been some recent work (and not so recent work) on *huge pages*, which seem to have settled on 2MB huge pages. In either case, with 48-bit addresses, the OS requires large page tables, regardless of whether they use regular or huge pages.